# EEU4C21/CSP55031/EEP55C26:
# Open Reconfigurable Networks

## Introduction to SDR

# What is an SDR?

- The term “Software-defined Radio” was first coined by Joseph Minolta in 1971
    - A modern communication system/platform that can adapt to operational environment
- First `publicly funded’ development was for the US Military - *SpeakEasy*
    - Use microprocessors to implement a large number of military communication standards
    - Support transmission and reception over a large range of frequencies (2 MHz to 2 GHz)
    - Adaptable sub-blocks allowing different functional parts to be swapped out in the field - for instance, modulation blocks or coding schemes to be altered
        - *SpeakEasy II* incorporated FPGA modules to implement digital baseband, the prototype system small enough to be carried in a truck
- Modern variants have condensed the same features to portable form-factors

# IEEE P1900.1 WG Terminologies

This working group created the following definitions to ensure common terminologies in the wireless communication space

- Radio - System or technology for wirelessly transmitting/receiving EM waves for information exchange
- Software - Modifiable instructions executed by a programmable processing device
- Physical Layer - the layer within the wireless protocol responsible for RF, IF and baseband processing, including channel coding [lowest layer of ISO model, adapted for wireless communication]
- Software Controlled - uses software processing within the radio system/device to *select parameters of operation*
- Software Defined - use of software processing within the radio system/device to *implement operating functions* (but not control)
- Software-controlled Radio - Radio system where some/all physical layer functions are software controlled
- Software-defined Radio - Radio system where some/all physical layer functions are software defined

# Modern SDR

- A class of reconfigurable/reprogrammable radios whose physical layer characteristics can be significantly modified via software changes
    - Capable of implementing different baseband functions at different times on the same platform
        - e.g., different modulation, error correction coding schemes
    - Possesses some software control over RF front-end operations,
        - e.g., transmission carrier frequency
- Typically, different baseband radio functionality are stored in memory
    - Different types of modulation, error correction coding, and other functional blocks are available to the SDR platform
    - Functional blocks can *potentially* be changed in real-time
- Operating parameters of functional blocks can be adjusted either through human intervention or through an automated process (cognitive systems)

# Key features[1]

- Multi-functionality – Possessing the ability to support multiple types of radio functions using the same digital communication system platform

- Global Mobility – Transparent operation with different communication networks located in different parts of the world, i.e., not confined to just one standard

- Compactness and Power Efficiency – Many communication standards can be supported with just one SDR platform

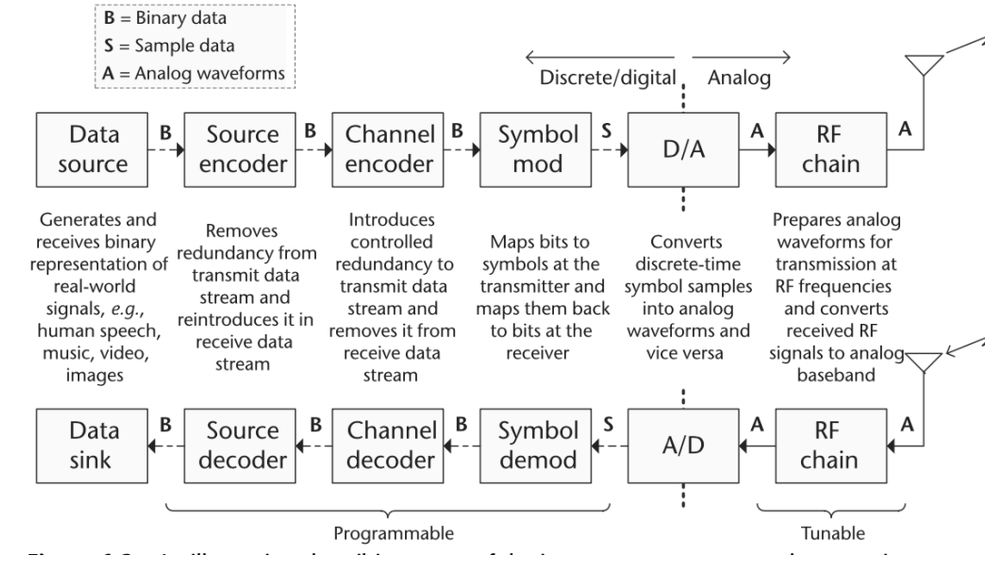
- Cost/Time to market – Baseband functions implemented as a software function not a [fixed] hardware block

- Ease of Upgrading – Firmware updates can be performed on SDR platform to enable functionality with latest communication standards

[1] See: Software Radio: A Modern Approach to Radio Engineering by Jeffrey H. Reed (Prentice Hall, 2002.)

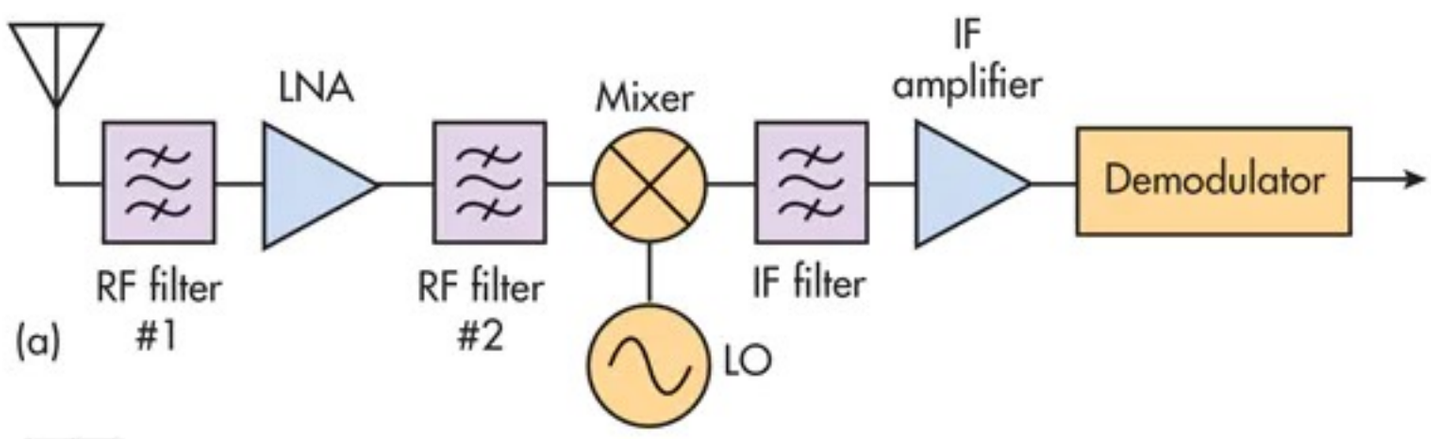
# Analogue-Digital domains in SDR Transceiver

- Baseband operations are more suitable for digital implementation
    - Using software on GPP or FPGA/DSPs for higher throughput
        - Can integrate real-time baseband processing for applications like spectrum sensing though hardware assisted software functions (a.l.a FPGAs)
- RF-chain involves up/down conversion, gain stages and antenna interfaces, which are still too complex for digital domain
    - Tunable RF components are the key factors
    - Move to Zero Intermediate Frequency style receivers



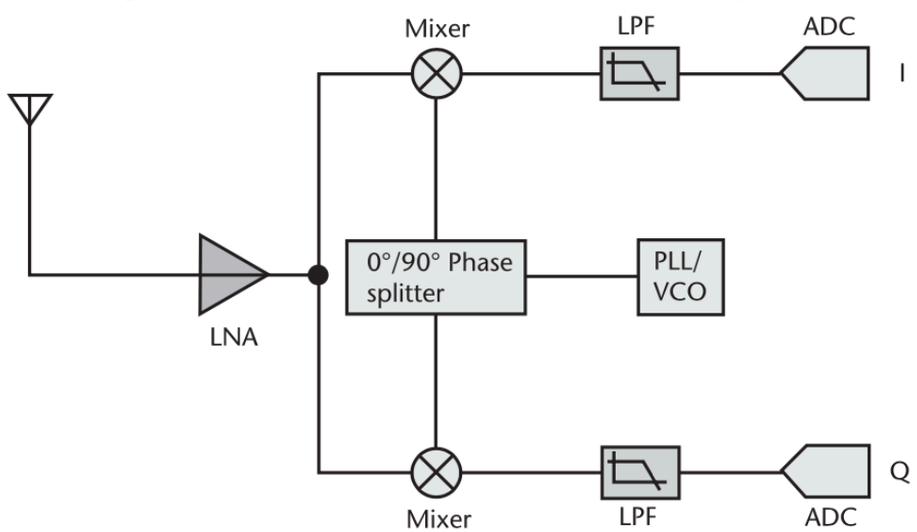
# RF blocks in SDR

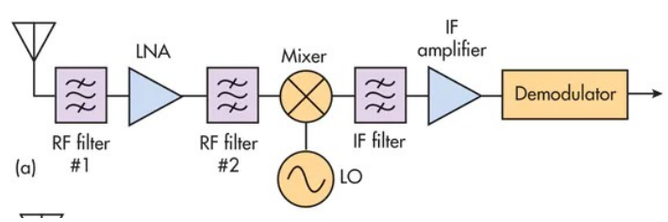
- Super-heterodyne Receiver has been the backbone of radio systems
    - Enables superior performance by utilising an Intermediate Frequency stage in the up and down-conversion chain
        - More precise control over filter designs, bandwidth, selectivity at the IF stage
        - Enables accurate frequency planning and gain distribution across stages
    - The high-performance RF/IF filters are large custom designs, increases the cost; also IF filters they determine the analogue bandwidth of the system, restricting the usability
    - However, it was mandated due to poor performance of alternative schemes (like direct downconversion and zero-IF)



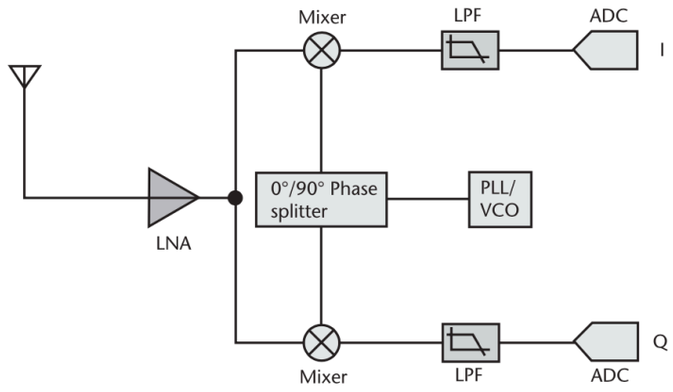
# RF blocks in SDR

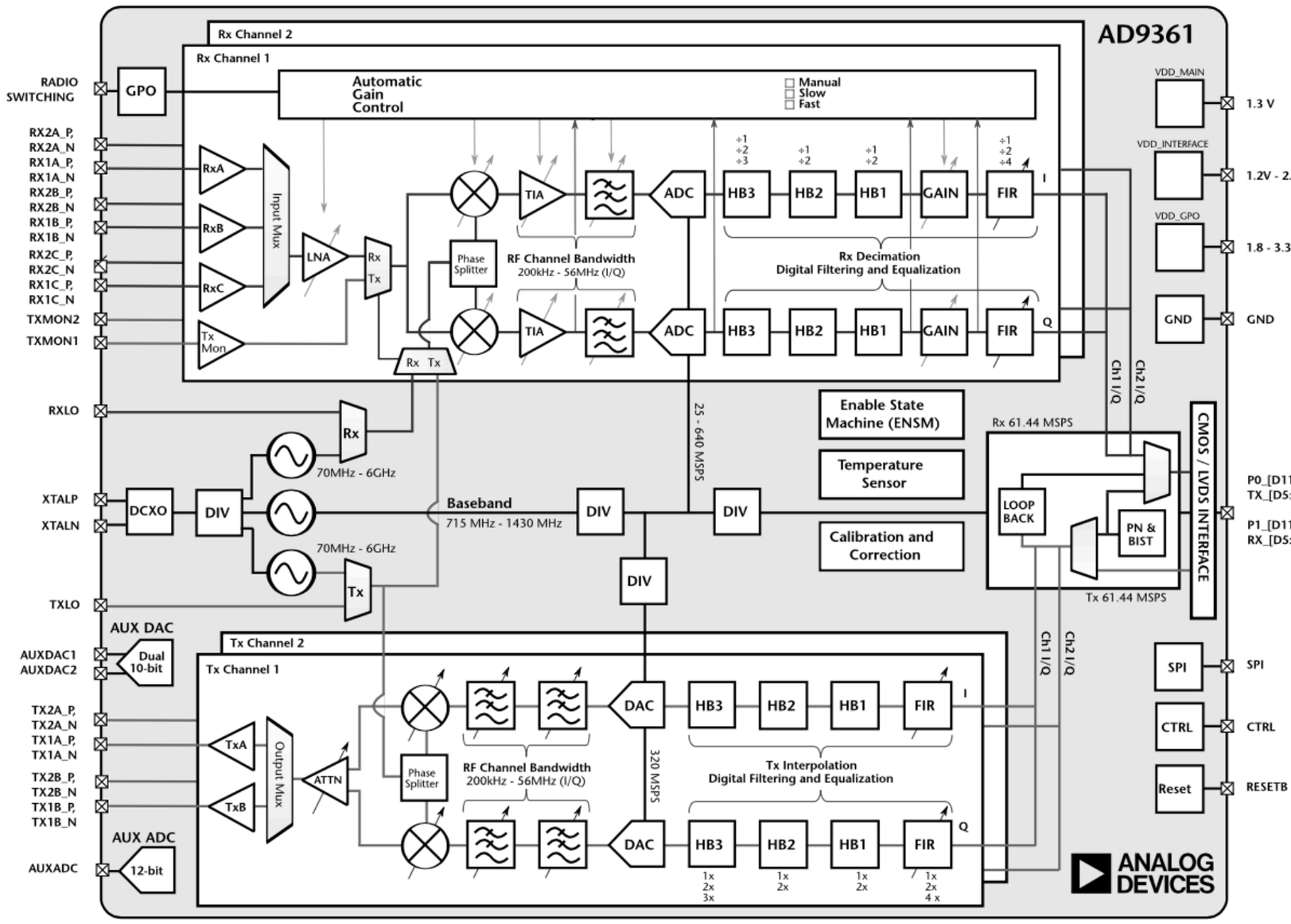
- Zero-IF (ZIF) architecture has re-emerged as a feasible solution in recent years
    - Use a single mixer and local oscillator set directly to the frequency band of interest to bring the signal down to baseband directly (as I and Q symbols/signals)
    - Moves filtering to digital domain and are mostly implemented in baseband; ADCs/DACs operating at baseband frequencies saving power
    - Prior issues of ZIF like carrier leakage and image frequency component due to imperfect mixing stage, process variations leading to poor transceiver performance addressed through integration into a single silicon stage
        - Lowering the power consumption and advances in transistor/IC designs enabled integration
    - Additional (digital) signal processing capabilities also enables unparalleled improvements in the RF-baseband interface
- Modern SDRs are enabled by such configurable RF front end devices that integrate RF, Analogue & Digital signal chain into a single CMOS device (like the Analog Devices AD9361 in PLUTO SDR)



# RF blocks in SDR



# Baseband (processing) architectures

**General Purpose Microprocessor**

- Highly flexible, easy to implement new software-driven functions
- Could struggle with complex mathematical operations and algorithms (signal processing type)
- Power inefficient in many scenarios

**Digital Signal Processor (DSP)**

- Specialised for performing signal-processing style computations, with processor-style operation - easy to implement functions
- Limited parallelism, hence slow for computationally expensive tasks
- Very efficient for certain signal processing tasks

**Field Programmable Gate Array (FPGA)**

- Fully flexible digital canvas to extract most performance through hardware; however specialised design process, so not as easy to design as pure software style functions.
    - Hybrid FPGA architectures can improve flexibility (e.g., Xilinx Zynq platform)
- Energy efficiency dependent on the application

# SDR Hardware

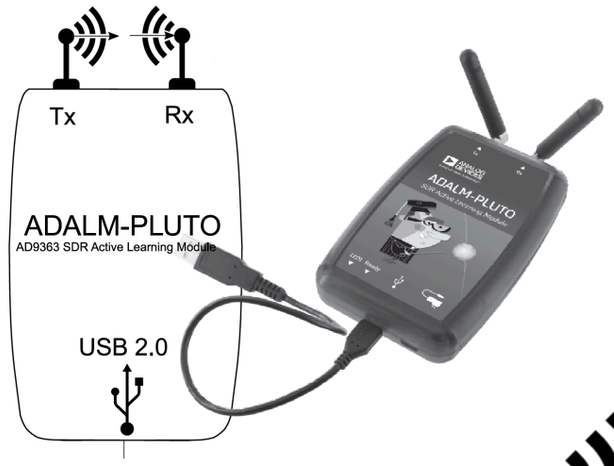
Recall that an SDR device is one radio node in a communication system. The system could be

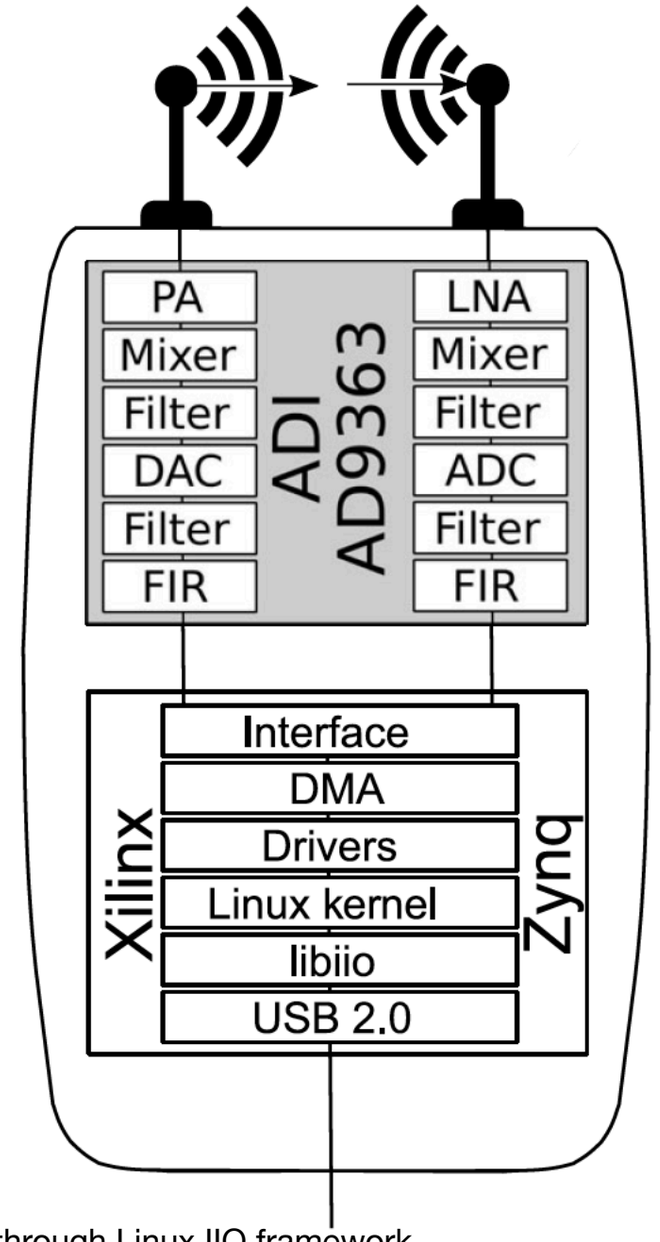
- A simple two-node point to point link
- A one-way broadcast network
- A large multi-hop ad-hoc network
- And anything else in between ….

Our Pluto SDR + MATLAB combination can be a single node in this larger communication system

- Might require additional external components
  - Specialised external filters
  - Band-specific antennas

# Pluto SDR



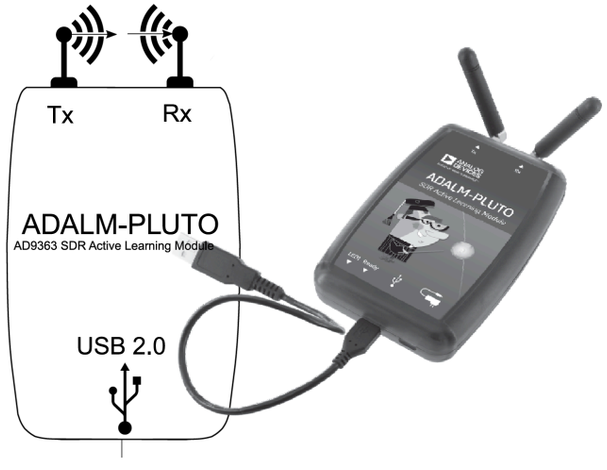




Pluto is a low-cost platform for SDR experimentation and education, composed of:

- An analog RF section
    - Antenna IF, RF Filters, MUX, LNA, Mixer …
        - Specialised Antennas and RF filters to be added if need be
    - Implemented on a single chip AD9363 device
- An analog baseband section
    - Programmable analog filters, ADCs and DACs
    - Also integrated in the AD9363
- Digital Signal Processing blocks
    - Includes fixed filters, used defined filters, other processing functions
    - Split between AD9363, Programmable Logic and Software backend
        - Fixed filters (decimation & interpolation filters) and programmable FIR filters - AD9363
        - Baseband processing that can be deployed on the Xilinx Zynq FPGA fabric
        - MATLAB-Simulink processing blocks (for 4C21)
- The Zynq FPGA provides interfaces to the programmable blocks of the SDR platform from MATLAB/Simulink through Linux IIO framework

More details of the individual blocks in section 5.1.x of the text

# Pluto SDR

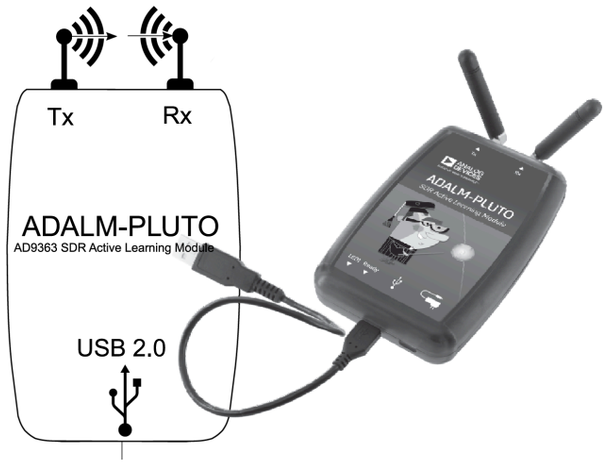



Specifications:

- RF Transmit/Receive
    - 300 Hz to 3.8 GHz at 200 Hz to 20 kHz channel bandwidth
    - 12-bit ADC and DAC
    - Local Oscillator - 2.4 Hz step size; Sample Rate - 5 Hz step size
- USB Interface
    - USB 2.0 - device mode with libiio - transfers I-Q data to host
        - Can be configured as a network device, USB serial device or Mass storage device
    - USB 2.0 - host mode
        - Mass storage, Wired/Wireless LAN or remote access modes
        - Applicable when managing SDRs through SDN approach (more details in SDN section)
- External power
    - powers the device when used in host-mode

NOTE: You could configure the device to cover a larger transceiver frequency range

# Pluto SDR

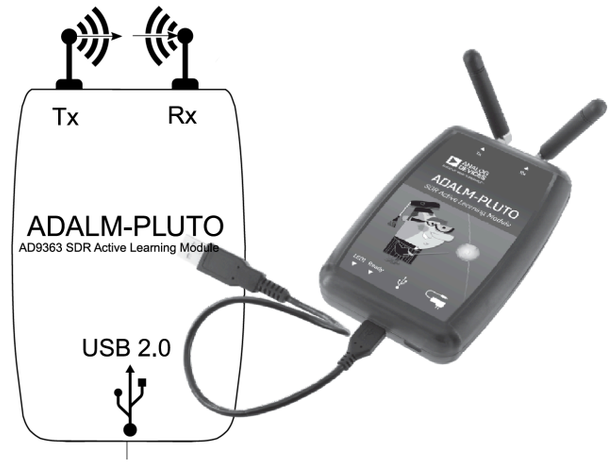


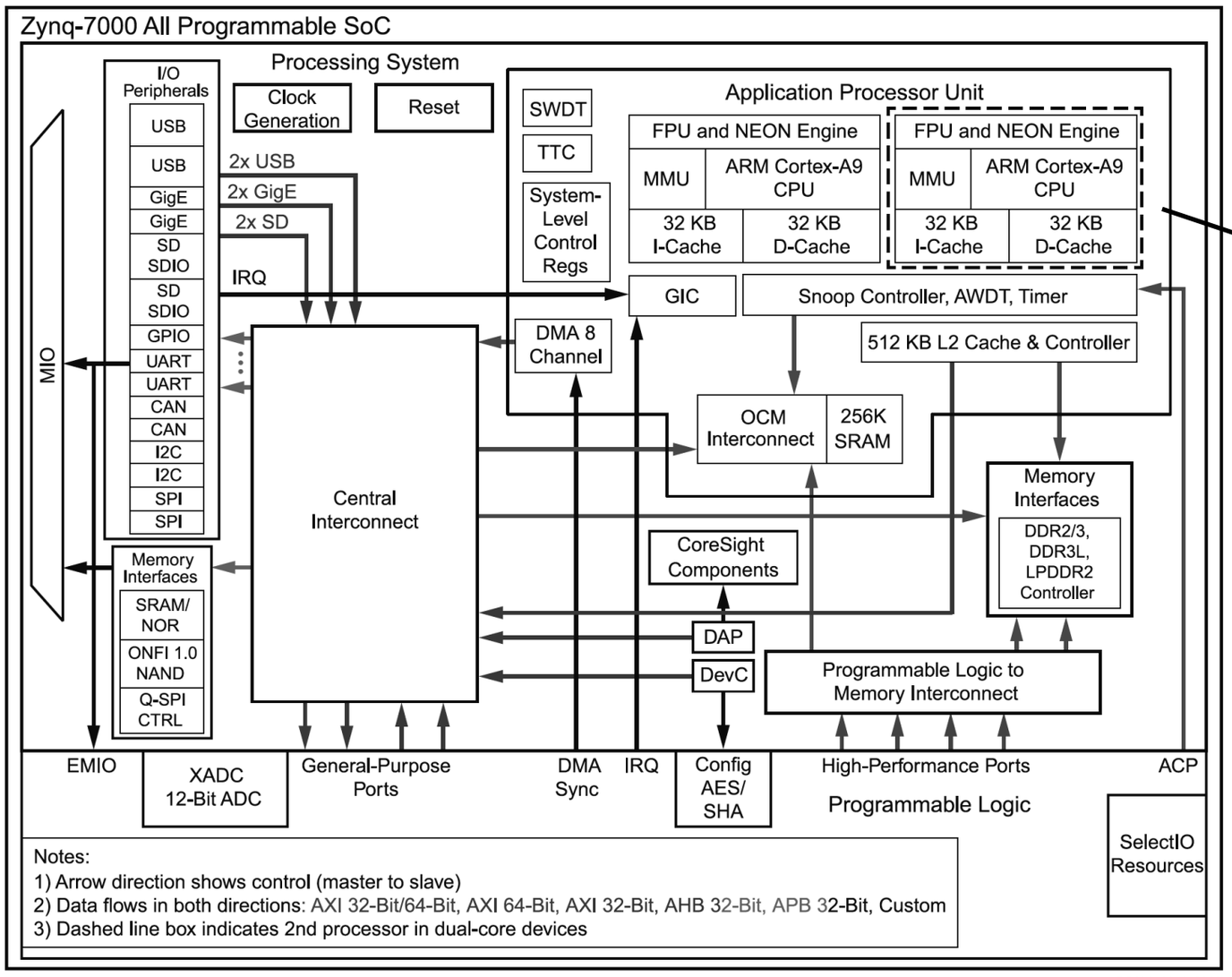

Zynq-7000 and Linux IIO

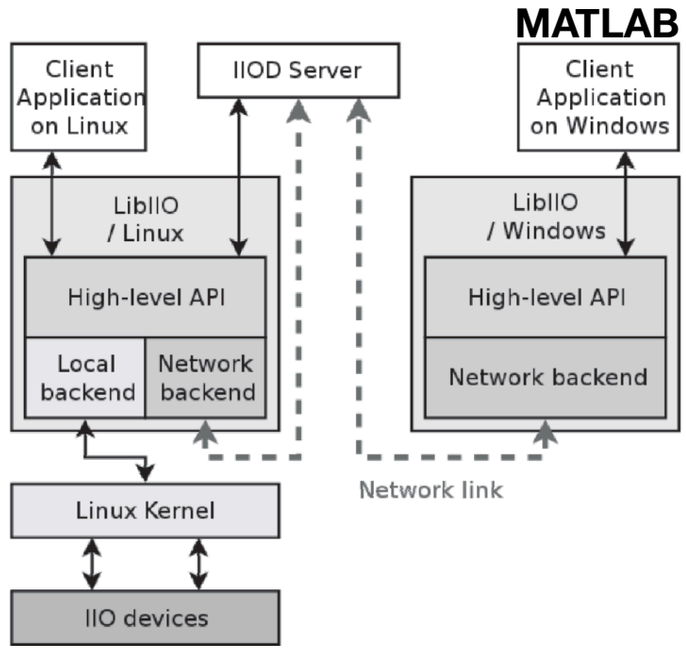
- The ARM cores on the Zynq platform runs a linux kernel that provides a standard interface to the host
    - Seamless movement of I/Q data samples to most Windows/Linux/MacOS devices
- In addition, the FPGA region implements a high speed decimation filter to extend the sampling rate of the RF front end
- The Linux kernel provides simple user interface to AD9363s programmable parameters
    - IIO-Driver within the linux kernel on the ARM cores
    - User-space library for accessing resources within the memory mapped space (both ARM and host-end)
    - The IIO-Daemon fo remote connection to the client running on ARM core
    - For 4C21 labs, we are using MATLAB/Simulink as the IIO Client on our host machine (Windows)

# Pluto SDR

## Zynq-7000 and Linux IIO



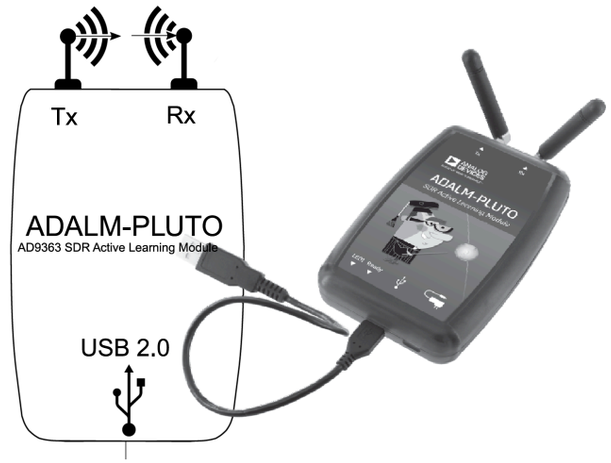
# Using MATLAB as IIO client

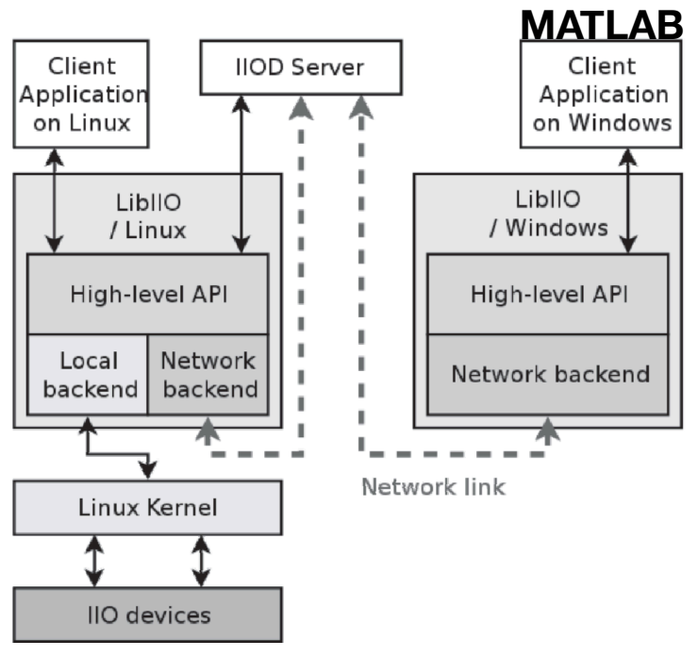
MATLAB - a cross platform IIO client to configure and work with PLUTO

- Uses a Pluto system object interface via the COMMs toolbox and hardware support package
    - With the object interface, you have a set of standard functions
    - Think of them as APIs - pass values to them to read/write data and to configure parameters
    - More advanced configurations can deploy new FPGA bitstreams for complex baseband models and high-bit rate communication (not covered as it could brick  the SDR if done wrongly)
- Two basic system objects

```
comm.SDRRxPluto

comm.SDRTxPluto
```



# Using MATLAB as IIO client

```
>> rx = sdrrx('Pluto');
>> tx = sdrtx('Pluto');
```

Instantiates a transmitter and receiver object;

Note that the object here is instantiated directly.

```
>> rx
rx =
  comm.SDRRxPluto with properties:
 Main
                DeviceName: 'Pluto'
                   RadioID: 'usb:0'
           CenterFrequency: 2.4000e+09
                GainSource: 'AGC Slow Attack'
            ChannelMapping: 1
         BasebandSampleRate: 1000000
            OutputDataType: 'int16'
           SamplesPerFrame: 20000
           EnableBurstMode: false
  ShowAdvancedProperties: false
Show all properties
```

You can use the object to see the list the default settings of the specific path.

Here we see the parameters for the RX path

Try these on the MATLAB console in the LAB

# Using MATLAB as IIO client

The parameters specify attributes of the device collected through the system object

*DeviceName* - Pluto is used for Pluto SDR; it is one among the many SDR devices in its family

*RadioID* - enumerates the interface used to connect to the host - USB:x for our use case, but could be others (like ip:<ip-address> if using the Ethernet interface)

*CenterFrequency* - specifies the current RF centre frequency in Hz. These could be different for Tx and Rx interfaces (use the Tx object to see the Tx interface)

*BasebandSampleRate* - defines the sample rate of the I/Q chains. This will be identical across Rx and Tx due to the shared clock generation logic for ADC & DAC

# Using MATLAB as IIO client

*GainSource* - specifies how the gain value for the internal chain is specified. This controls multiple stages within the rx path in the AD9363

*ChannelMapping* - specifies the channel to transmit/receive - for Pluto, there is only a single channel configuration; however in other SDRs that support MIMO, this parameter has significance

*OutputDataType* - specifies the data format used when reading from the device. The ADC/DAC has 12-bit resolution, but the interface outputs sign extended 16-bit integer type values by default.

*SamplesPerFrame* - specifies the size of the frame buffer and in turn the number of samples in the frame read from the device. Note that values will be contiguous unless there was an overflow in the buffer.

NOTE: some of these parameters only make sense in the receiver context, so in case of transmitter, not all the above parameters will be seen

# Using MATLAB as IIO client

So how to use MATLAB to receive some data?

```
1 %% Assumes frameSize, framesToCollect are defined
2 % Perform data collection then offline processing
3 data = zeros(frameSize, framesToCollect);
4 % Collect all frames in continuity
5 for frame = 1:framesToCollect
6    [d,valid,of] = rx();
7    % Collect data without overflow and is valid
8    if ~valid
9       warning('Data invalid')
10   elseif of
11      warning('Overflow occurred')
12   else
13      data(:,frame) = d;
14   end
15 end
16
17 % Process data once received
18 sa1 = dsp.SpectrumAnalyzer;
19 for frame = 1:framesToCollect
20    sa1(data(:,frame)); % Algorithm processing
21 end
```

Initialise a data buffer with 0's

Loop through and read as many frames

Each frame data also comes with data flags

Check flag status before saving the data to the buffer

Once data is available, split them into frames as needed

Example here shows the use of a spectrum analyser tool

# Using MATLAB as IIO client

So how to use MATLAB to receive some data?

```
23 % Save data for processing
24 bfw = comm.BasebandFileWriter('PlutoData.bb',...
25     rx.BasebandSampleRate,rx.CenterFrequency);
26 % Save data as a column
27 bfw(data(:));
28 bfw.release();
```

You could save the data using the `BasebandFileWriter( )` function to a file in your system for later processing

```
1 % Load data and perform processing
2 bfr = comm.BasebandFileReader(bfw.Filename,...
3    'SamplesPerFrame',frameSize);
4 sa2 = dsp.SpectrumAnalyzer;
5 % Process each frame from the saved file
6 for frame = 1:framesToCollect
7 sa2(bfr()); % Algorithm processing
8 end
```

Saved data can be read back at a later phase to do the processing. The function `BasebandFileReader( )` will read and reorganise the contents of the file to mimic the received buffer content

# Using MATLAB as IIO client

Once you tune your processing algorithm, this could be placed back into the main loop to do *streaming* processing.

```
1 %% Add the algorithm into the main loop
2 sa3 = dsp.SpectrumAnalyzer;
3 % Process each frame immediately
4 for frame = 1:framesToCollect
5     [d,valid,of] = rx();
6     % Check overflow and valid flags before
processing
7     if ~valid
8         warning('Data invalid')
9     elseif of
10        warning('Overflow occurred')
11    else
12        sa3(d); % Your algorithm steps here
13    end
14 end
```

# Using MATLAB as IIO client

What if you want to transmit?

```
1 %% Example to transmit all zeros
2 tx = sdrtx('Pluto');
3 % Specify frequency of modulating and carrier
4 fs = 1e6; fc = 1e4;
5 s = 2*pi*fs*fc*(1:2^14).6';  % Vector for transmission
7 wave = complex(cos(s), sin(s)); % I-Q components
8 tx.trasmitRepeat(wave);
```

NOTE: The transceiver is designed in such a way that whenever Pluto is powered on, the transmitter will transmit data in its buffer repeatedly. Hence, even though you are using only the receiver object, there is some data transmitted by the device (which could cause interference if both Rx and Tx LO frequencies are the same). The above code is useful in this case, forcing the Tx to transmit 0s and reduce the transmit energy - however, there can still be some power leakage into the receiver's data.  The solution in this case is to set the transmitter CenterFrequency away from the receiver using something along the lines of:

```
tx.CenterFrequency = rx.CenterFrequency + 100e6;
```

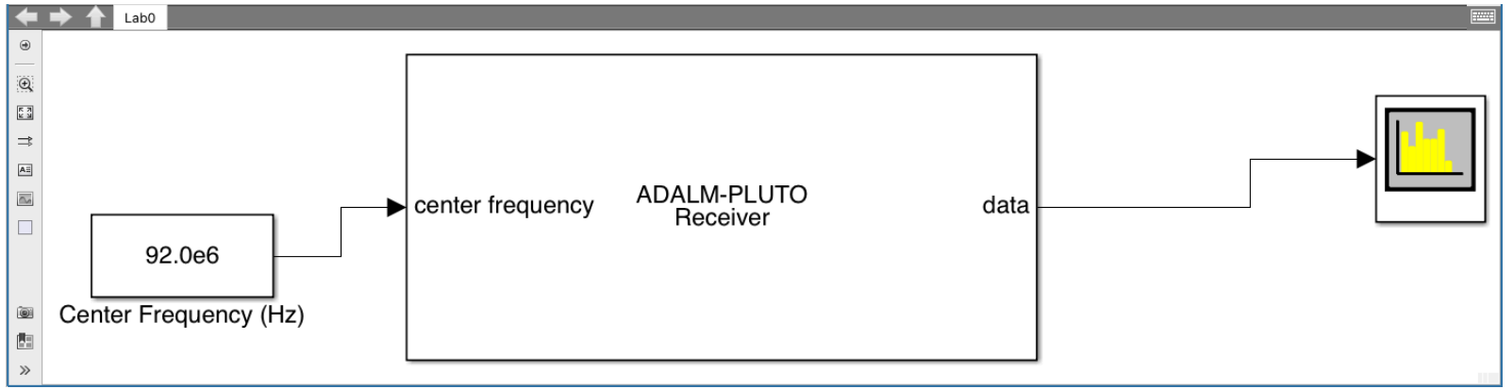
# Using MATLAB as IIO client

An example - sense the spectrum in this vicinity

```
1 % Check the radio spectrum in the default configuration
2 rx = sdrrx('Pluto');
3 rx.SamplesPerFrame = 2^15;
4 sa = dsp.SpectrumAnalyzer;
5 sa.SampleRate = rx.BasebandSampleRate;
6 for k=1:1e3
7     sa(rx());
8 end
```

# Using Simulink as IIO client

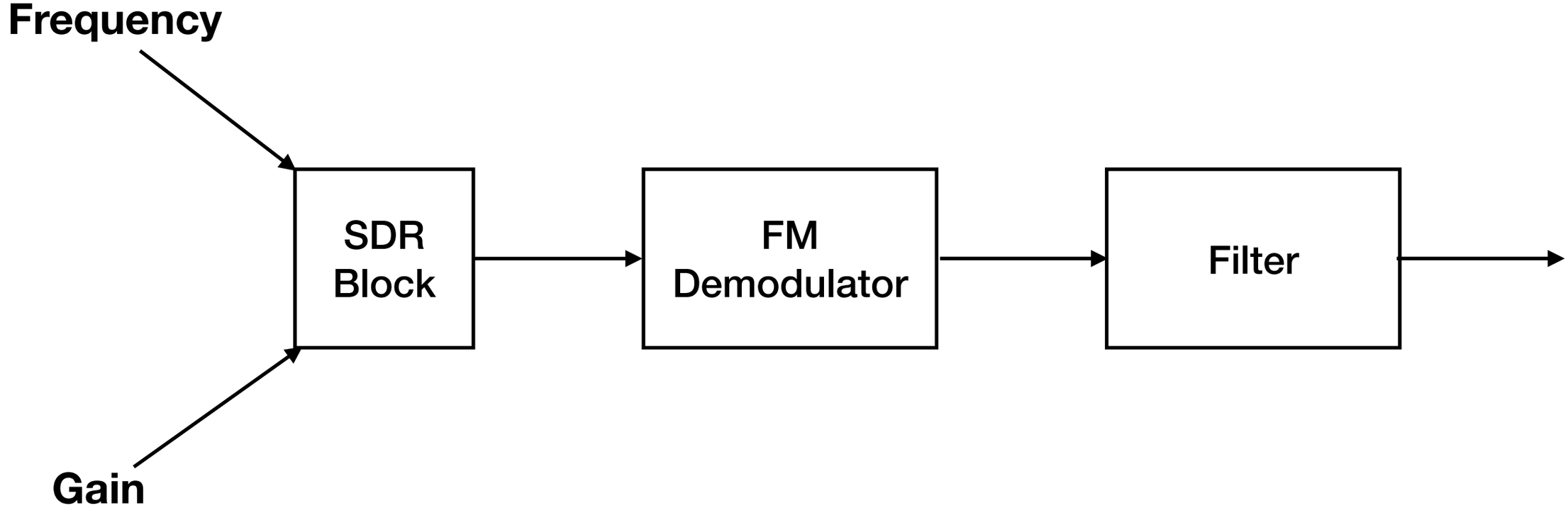
Simulink allows the processing blocks to be built using predefined components




- Drag and drop the PLUTO receiver block which instantiates the system objects
- Like with MATLAB commands, the properties of this block can be set to specify the same parameters
- More abstracted way to build radio processing blocks
- Toolboxes could also be instantiated in the same way - the figure shows our spectrum analyser using Simulink blocks

# Using Simulink as IIO client

Let us build an FM Radio



# Aside: Latency and Delays

Abstracting details allows the radio elements to be more configurable and user friendly;

However, the number of layers involved does create a delay chain;

At some point, you will start observing the delays in transmitted and received data;

There are both deterministic and non-deterministic delays caused by the different layers and stacks